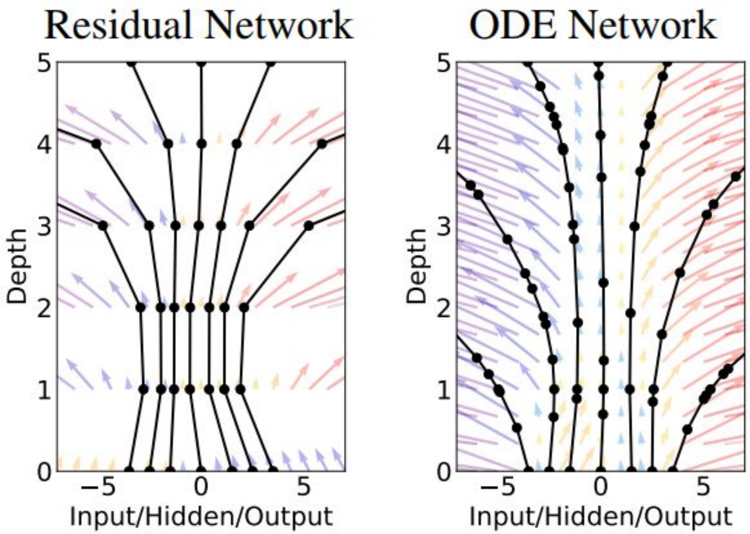
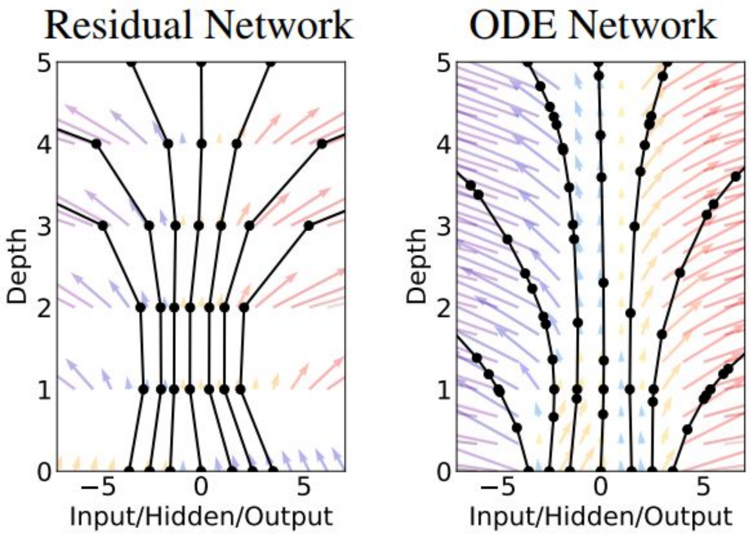
ShelfWise

Empowering Retail with Image Recognition

www.shelfwise.ai



Neural Ordinary Differential Equations



Neural Ordinary Differential Equations

Best paper award at NeurIPS 2018

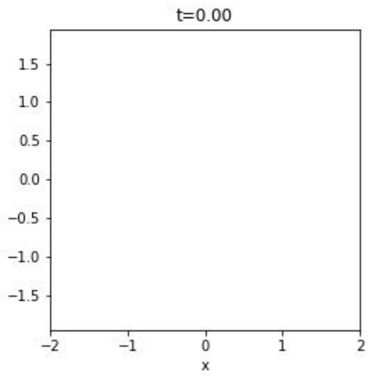
Content

- Motivation for using differential equations
- Simple implementation of black-box solver (in python)
- Integrating NN with solvers
- Computing gradients through ODE - adjoint method
- Results and potential applications
- **Appendix**: Continuous Normalizing Flows

Solving dynamical systems with ODEs

Motivations for description of dynamical systems

Time evolution of states in some unknown system



Assuming full knowledge about the problem

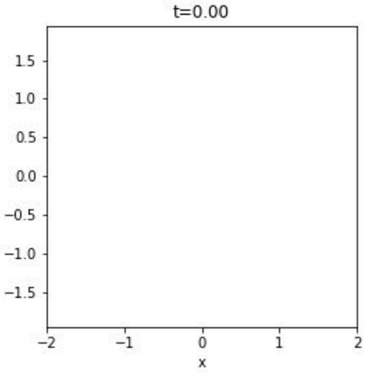
clue: a problem of three bodies connected with springs

How many parameters (at most) we have to know to fully describe this problem and be able to infer future states ?

3 bodies * 2 (initial position) * 2 (initial velocity) + 3 (spring strength) + 3 (spring equilibrium distance)

Motivations for description of dynamical systems

States evolution in some unknown system



Now, try to solve this problem with regular Neural Network …

Hidden state transformation in NNs

- In **regular** neural networks (NNs) states are transformed by series of discrete transformations

$$\mathbf{h}_{t+1} = f(\mathbf{h}_t)$$

- where f is e.g. some **Dense** or **Convolutional layer**
- t (a layer index) can be interpreted as time index, such that we transform some input data at t=0 to the output space at t=N
- in order to learn some dynamical system (e.g. physical system) with RNNs we must discretize time steps
- there are problems where expressing time as a continuous variable is more natural e.g. physical simulations, events which happen at irregular intervals
- so it may be profitable to express our problem as a **differential equation**

ODE and initial value problem

- We are interested in problems described by following ordinary differential equation (**ODE**):

$$\frac{d\mathbf{h}(t)}{dt} = g(t, \mathbf{h}(t))$$

- with some known initial value for **h**: **h**(t=0) = $\mathbf{h}_0$
- this kind of equation can describe plenty of dynamical (e.g. physical) systems
- **For example:** radioactive decay (with N being number of atoms/mass of radioactive material):

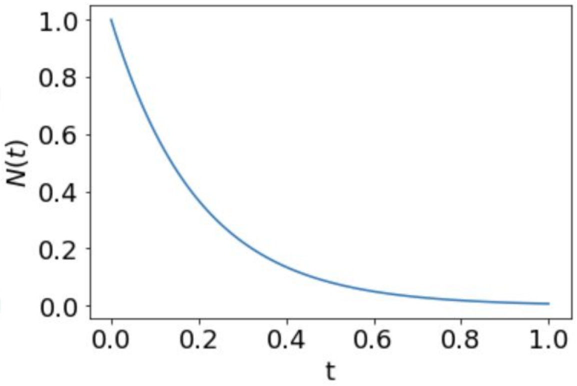
$$\frac{dN(t)}{dt} = -\lambda N(t)$$

ODE case study: Radioactive decay - differential form

- **For example:** radioactive decay (with N being number of atoms/mass of radioactive material):

$$\frac{dN(t)}{dt} = -\lambda N(t)$$

- Analytical solution: $N(t) = N_0 e^{-\lambda t}$




```python
t = np.linspace(0, 1, 100)
λ = 5.0
N = lambda t: np.exp(-λ*t)
plt.plot(t, N(t))
```

ODE case study: Radioactive decay - integral form

- Any differential equation of <u>form</u> can be expressed in terms of <u>integral</u>

$$\frac{d\mathbf{h}(t)}{dt} = g(t, \mathbf{h}(t))$$

$$\mathbf{h}(t) = \mathbf{h}(0) + \int_0^t dt' g(t', \mathbf{h}(t'))$$

ODE case study: Radioactive decay - numerical integration

- Integral form of ODE:

$$\mathbf{h}(t) = \mathbf{h}(0) + \int_0^t dt' g(t', \mathbf{h}(t'))$$

solution at time t depends on all values at t' < t

- We are going to use methods which will approximate the integral
- The simplest approximation replaces integral with simple sum taking time at discrete time steps:

$$\mathbf{h}(N\Delta t) = \mathbf{h}(0) + \Delta t \sum_{k=0}^{N-1} g(k\Delta t, \mathbf{h}(k\Delta t))$$

- Using shorter notation we get:

$$\mathbf{h}_N = \mathbf{h}_0 + \Delta t \sum_{k=0}^{N-1} g(k\Delta t, \mathbf{h}_k)$$

ODE case study: Radioactive decay - numerical integration

- Using shorter notation we get:

$$\mathbf{h}_N = \mathbf{h}_0 + \Delta t \sum_{k=0}^{N-1} g\left(k\Delta t, \mathbf{h}_k\right)$$

$$= \underbrace{\boxed{\left(\mathbf{h}_0 + \Delta t \sum_{k=0}^{N-2} g\left(k\Delta t, \mathbf{h}_k\right)\right)}}_{\mathbf{h}_{N-1}} + \Delta t g\left(\left(N-1\right)\Delta t, \mathbf{h}_{N-1}\right)$$

- From which we obtain following recurrence formula:

$$\mathbf{h}_N = \mathbf{h}_{N-1} + \Delta t g\left(\left(N-1\right)\Delta t, \mathbf{h}_{N-1}\right)$$

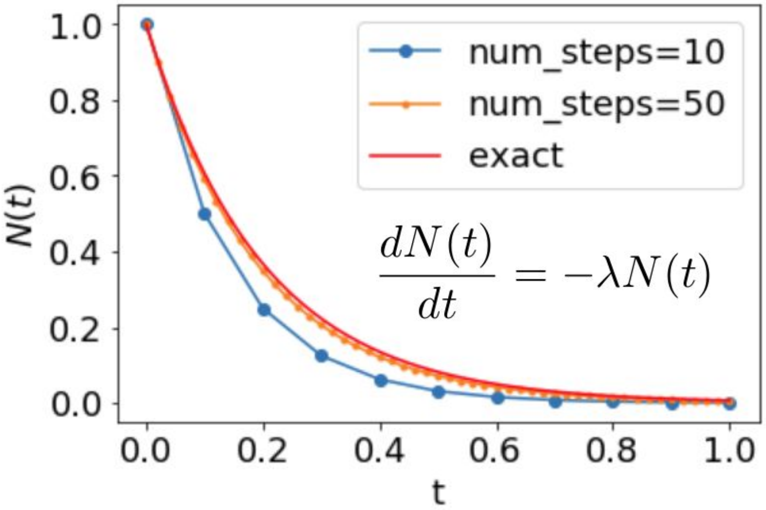
Euler's method for solving differential equations.
Same result can be obtained by approximating derivative
in the original ODE problem

ODE case study: Radioactive decay - numerical integration

- **Euler** method is the simplest method known:

$$\mathbf{h}_N = \mathbf{h}_{N-1} + \Delta t g\left(\left(N-1\right)\Delta t, \mathbf{h}_{N-1}\right)$$

```
hist = []
# fixed grid
num_steps = 10
Δt = 1/num_steps
# initial condition
Nk = 1.0;
tk  = 0.0
# integration
hist.append((tk, Nk))
for k in range(num_steps):
    Nk = Nk - λ * Δt * Nk
    tk  = tk + Δt
    hist.append((tk, Nk))
```



ODE case study: Towards black box solvers

- Lets refactor this code a bit and create more general solver:

$$\mathbf{h}(t) = \mathbf{h}(0) + \int_0^t dt' \boxed{g(t', \mathbf{h}(t'))}$$

$$\mathbf{h}_N = \mathbf{h}_{N-1} + \Delta t g((N-1)\Delta t, \mathbf{h}_{N-1})$$

```python
def g_fn(tk, hk):
    return - λ * hk

def euler_step(dt, tk, hk, fun):
    return hk + dt * fun(tk, hk)
```

```python
hist = []
# fixed grid
num_steps = 10
Δt = 1/num_steps
# initial condition
Nk = 1.0;
tk  = 0.0
# integration
hist.append((tk, Nk))
for k in range(num_steps):
    Nk = Nk - λ * Δt * Nk
    tk  = tk + Δt
    hist.append((tk, Nk))
```

```python
hist = []
# fixed grid
num_steps = 50
Δt = 1/num_steps
# initial condition
Nk = 1.0;
tk  = 0.0
# integration
hist.append((tk, Nk))
for k in range(num_steps):
    Nk = euler_step(Δt, tk, Nk, g_fn)
    tk  = tk + Δt
    hist.append((tk, Nk))
```

integration of time variable

ODE case study: Towards black box solvers

- Implementation of more general solver
- Here we assume fixed grid approach, adaptive methods are possible

```python
def g_fn(tk, hk):
    return - λ * hk

def euler_step(dt, tk, hk, fun):
    return hk + dt * fun(tk, hk)
```

```python
hist = []
# fixed grid
num_steps = 50
Δt = 1/num_steps
# initial condition
Nk = 1.0;
tk  = 0.0
# integration
hist.append((tk, Nk))
for k in range(num_steps):
    Nk = euler_step(Δt, tk, Nk, g_fn)
    tk  = tk + Δt
    hist.append((tk, Nk))
```

We would like to have:

- problem function, can be easily changed
- example solver, can be easily changed

A general fixed grid solver in 10 lines

```python
def odeint(func, y0, t, solver):
    Δts = t[1:] - t[:-1]
    tk = t[0]
    yk = y0
    hist = [(tk, y0)]
    for Δt in Δts:
        yk = solver(Δt, tk, yk, func)
        tk = tk + Δt
        hist.append((tk, yk))
    return hist
```

```python
hist = odeint(
    g_fn, 1.0, np.linspace(0, 1, 50),
    solver=euler_step
)
```

ODE case study: Towards black box solvers

- Implementing better solvers we can achieve better accuracy with less steps

```python
def odeint(func, y0, t, solver):
    Δts = t[1:] - t[:-1]
    tk = t[0]
    yk = y0
    hist = [(tk, y0)]
    for Δt in Δts:
        yk = solver(Δt, tk, yk, func)
        tk = tk + Δt
        hist.append((tk, yk))
    return hist
```

- Euler method (explicit version)

```python
def euler_step(dt, tk, hk, fun):
    return hk + dt * fun(tk, hk)
```

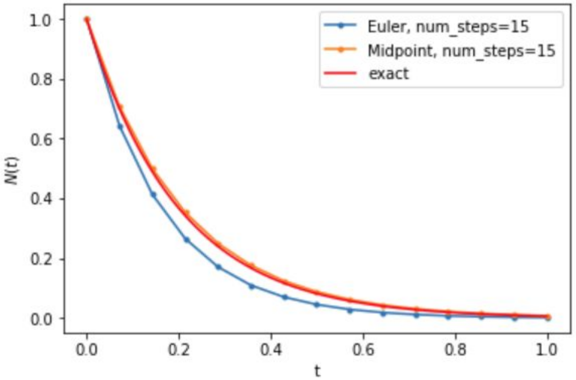
- Midpoint method (or RK2) - 2nd order method

```python
def midpoint_step(Δt, tk, hk, fun):
    k1 = fun(tk, hk)
    k2 = fun(tk + Δt, hk + Δt * k1)
    return hk + Δt * (k1 + k2) / 2
```

- Euler vs Midpoint

```python
t_grid = np.linspace(0, 1, 15)
hist_euler = odeint(g_fn, 1.0, t_grid, euler_step)
hist_midpoint = odeint(g_fn, 1.0, t_grid, midpoint_step)
```

- Midpoint has lower error (obviously) =>



ODE case study: Towards black box solvers

- **odeint** is a general purpose ODE solver, one must provide **fun(t, h_t)**, initial conditions, timesteps at which function will be evaluated and **solver**
- there are plenty of ready to use **black box** solvers:
  - fixed grid or adaptive methods
  - implicit or explicit methods
  - ...
- higher order methods like Runge–Kutta (RK4) or Adams–Bashforth guarantee better numerical accuracy
- all of them can be implemented in a common interface of form (e.g. scipy):

```
>>> def pend(y, t, b, c):
...     theta, omega = y
...     dydt = [omega, -b*omega - c*np.sin(theta)]
...     return dydt
```

```
>>> from scipy.integrate import odeint
>>> sol = odeint(pend, y0, t, args=(b, c))
```

- tf.contrib.integrate.odeint <- another example

Integrating Neural Networks with ODE solver

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$$

- **odeint** can be used to integrate response of some **Neural Network**

```
def odeint(func, y0, t, solver):
    Δts = t[1:] - t[:-1]
    tk = t[0]
    yk = y0
    hist = [(tk, y0)]
    for Δt in Δts:
        yk = solver(Δt, tk, yk, func)
        tk = tk + Δt
        hist.append((tk, yk))
    return hist
```

- Define some model with __call__ function

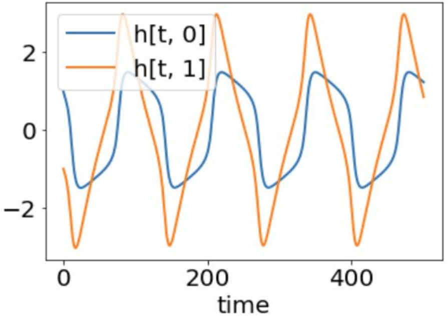
```
class Module(keras.Model):
    def __init__(self, nf):
        super(Module, self).__init__()
        self.dense_1 = Dense(nf, activation='tanh')
        self.dense_2 = Dense(nf, activation='tanh')

    def call(self, inputs, **kwargs):
        t, x = inputs
        h = self.dense_1(x)
        return self.dense_2(h) - 0.25 * x
```

- Integrate (small change in the solver definition)

```
t_grid = np.linspace(0, 500., 2000)
h0 = tf.to_float([[1.0, -1.0]])
model = Module(2)
hist = odeint(model, h0, t_grid, midpoint_step_keras)
```

```
def midpoint_step_keras(Δt, tk, hk, fun):
    k1 = fun([tk, hk])
    k2 = fun([tk + Δt, hk + Δt * k1])
    return hk + Δt * (k1 + k2) / 2
```

- Plot history:



Integrating Neural Networks - ResNet analogy

- Euler method is the simplest one:

$$\mathbf{h}_N = \mathbf{h}_{N-1} + \Delta t g\left(\left(N-1\right)\Delta t, \mathbf{h}_{N-1}\right)$$

- Some people find this equation similar to **ResNet** skip connection

$$\mathbf{h}_{l+1} = \mathbf{h}_l + \mathrm{NNetwork}(\mathbf{h}_l)$$

l - enumerates layers

- However, each step in **ResNet** has its own parameters, here are reusable!
- Nevertheless a possible connection between **NeuralODEs** and skip connections in an interesting question

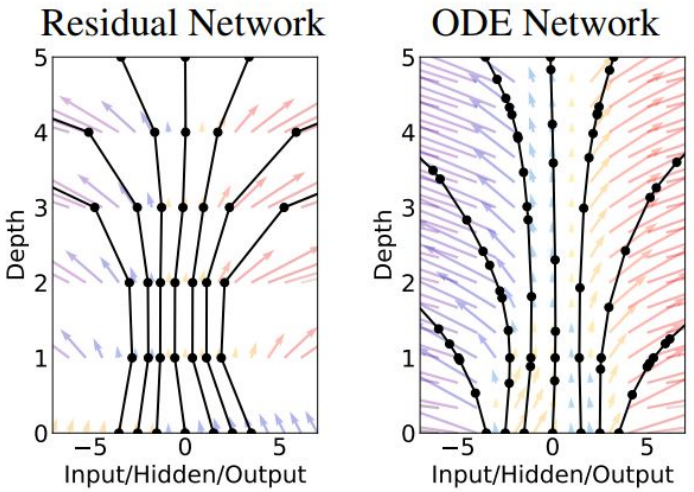


Figure 1: *Left:* A Residual network defines a discrete sequence of finite transformations. *Right:* A ODE network defines a vector field, which continuously transforms the state. *Both:* Circles represent evaluation locations.

Integrating Neural Networks - black box solvers

- We can use existing (and efficient) implementation of solvers to **integrate NNs** dynamics
- The **memory cost is O(1)**, due to **reversibility** i.e. we don’t need to store all activations in the graph, we can easily recover them by backward integration (i.e. time reversed integration)
- Complex dynamics can be modeled with fewer parameters
- We can control **accuracy/speed trade-off** with adaptive solvers by setting **lower/higher error tolerances**
- Hidden states can be accessed at any value of t - **no discrete time steps** as in RestNet skip connection

NeuralODE - forward mode integration summary

So far we have discussed:

- New type of NNs where ResNet-like skip connection

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta_t)$$

- is replaced by ODE (a new type of NN)

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$$

- ODE is then solved with black box solver accessed e.g. via **odeint** function
- Output state is then used to compute some loss:

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta) dt\right) = L\left(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)\right)$$

- This loss is then used to compute gradients, **but how to do it?**

Backpropagating through NeuralODEs - a naive approach

NeuralODE - naive backpropagation

When using Euler method to solve ODE $\frac{d\mathbf{h}(t)}{dt} = \mathrm{NNetwork}\left(\mathbf{h}(t), t, \theta\right)$

- solver update step can be written as ResNet skip connection block

$$\mathbf{h}_{l+1} = \mathbf{h}_l + \mathrm{NNetwork}(\mathbf{h}_l)$$

```python
def euler_step(dt, tk, hk, fun):
    return hk + dt * fun(tk, hk)
```

- after **k-steps** we get just regular ResNet architecture build from **k blocks**

$$\mathbf{h}_1 = \mathbf{h}_0 + \mathrm{NNetwork}(\mathbf{h}_0)$$

$$\cdots$$

$$\mathbf{h}_k = \mathbf{h}_{k-1} + \mathrm{NNetwork}(\mathbf{h}_{k-1})$$

```python
# 100 layers ResNet
t = np.linspace(0, 1, 100)
net = NNetwork() # block fn
x_batch, y_labels = get_batch()
with tf.GradientTape() as g:
    x_hist = odeint(
        func=net,
        y0=x_batch,
        t=t,
        solver=euler
    )
    loss = loss_fn(x_hist, y_labels)

grads = tape.gradient(
    loss, net.weights
)
optimizer.apply_gradients(
    zip(grads, net.weights)
)
```

- gradient of loss can be easily computed with existing methods

$$\frac{\partial \mathcal{L}}{\partial \theta} = \mathrm{loss}\left(\mathbf{h}_k, y_{\mathrm{targets}}; \theta\right)$$

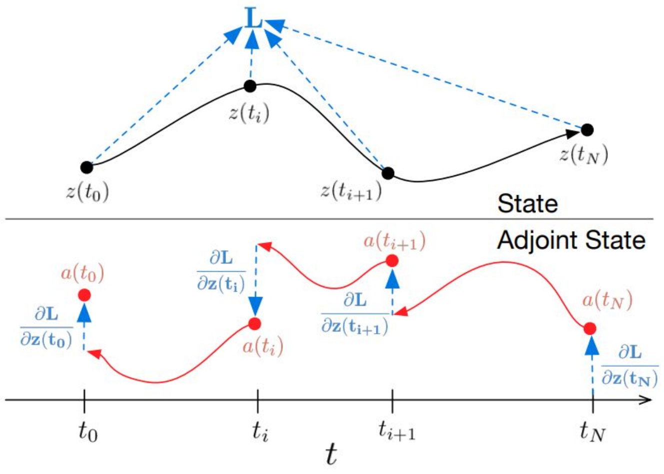
NeuralODE - naive backpropagation

- Many higher order solvers are also differentiable
- For example Midpoint (RK2), RK4 updates are easily differentiable

```
def midpoint_step(Δt, tk, hk, fun):
    k1 = fun(tk, hk)
    k2 = fun(tk + Δt, hk + Δt * k1)
    return hk + Δt * (k1 + k2) / 2
```

When naive backpropagation fails ?

- for example we want to integrate dynamical systems through **1M timesteps**, this would correspond to roughly 1M layer NN, so we will end up with memory issues,
- **memory issues arise** because we **need to store all activations** in the graph and higher order solvers even more activations,
- **backpropagation through adaptive solvers maybe infeasible** due to numerical errors, instability or just non-differentiability of the solver



Backpropagating through NeuralODEs - adjoint method

NeuralODE - adjoint method

- **Adjoint method** has been developed to overcome mentioned problems
- *Adjoint sensitivity method* has long history: (Pontryagin et al., **1962!**)
- **Adjoint method** can be understand as a continuous version of chain rule
- **Chain rule:** Consider following sequence of operations (*L* is a scalar loss):

$$\mathbf{h}_{t+1} = f\left(\mathbf{h}_t\right)$$
$$\mathcal{L} = \mathcal{L}\left(\mathbf{h}_{t+1}\right)$$

- We can compute gradient of L w.r.t input state using chain rule

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

- This formula is a core of any Deep Learning autograd implementation

NeuralODE - adjoint method (derivation)

- Consider following sequence of operations (L is a scalar loss)

$$\boxed{\mathbf{h}_{t+1}} = f(\mathbf{h}_t)$$
$$\mathcal{L} = \mathcal{L}(\mathbf{h}_{t+1})$$

- We can compute gradient of L w.r.t input state using chain rule

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

- We are interested in infinitesimal (continuous) change in hidden state:

$$\boxed{\mathbf{h}(t+\varepsilon)} = \mathbf{h}(t) + \int_t^{t+\varepsilon} f(\mathbf{h}(t'), t')\, dt' \quad \text{since} \quad \frac{\mathrm{d}\mathbf{h}(\mathrm{t})}{\mathrm{dt}} = \mathrm{f}(\mathbf{h}(\mathrm{t}), \mathrm{t})$$

- Same **chain rule** can be applied

$$\boxed{\frac{\partial L}{\partial \mathbf{h}(t)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}(t+\varepsilon)} \frac{\partial \mathbf{h}(t+\varepsilon)}{\partial \mathbf{h}(t)}$$

- **Adjoint state** is defined as:

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{h}(t)}$$

NeuralODE - adjoint method

- An infinitesimal change in the hidden state (1)

$$\mathbf{h}(t+\varepsilon) = \mathbf{h}(t) + \int_t^{t+\varepsilon} f(\mathbf{h}(t'), t')\, dt' \quad \text{since} \quad \frac{\mathrm{d}\mathbf{h}(\mathrm{t})}{\mathrm{dt}} = \mathrm{f}(\mathbf{h}(\mathrm{t}), \mathrm{t})$$

- Continuous **chain rule** can be applied (2): $\frac{\partial L}{\partial \mathbf{h}(t)} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}(t+\varepsilon)} \frac{\partial \mathbf{h}(t+\varepsilon)}{\partial \mathbf{h}(t)}$

- **Adjoint state** is defined as (3): $\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{h}(t)}$

- From Eq.(2) and Eq.(3) we get (4): $\mathbf{a}(t) = \mathbf{a}(t+\varepsilon) \frac{\partial \mathbf{h}(t+\varepsilon)}{\partial \mathbf{h}(t)}$

- By combining all equations above we can derive differential equation which describes dynamics of adjoint state:

$$\frac{d\mathbf{a}(t)}{dt} = \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} = -\mathbf{a}(t) \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)}$$

*should be **h** instead of **z***

full proof in paper

NeuralODE - adjoint method

- Final formulas:

$$\frac{\mathrm{d}\mathbf{h}(\mathrm{t})}{\mathrm{dt}} = \mathrm{f}(\mathbf{h}(\mathrm{t}), \mathrm{t})$$ forward dynamics

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{h}(t)}$$ adjoint state (definition)

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)\frac{\partial f(\mathbf{h}(t), t)}{\partial \mathbf{h}(t)}$$ adjoint state dynamics

- **Why adjoint state is important and useful ?**
  - When doing backpropagation we need to compute these two quantities $\frac{\partial L}{\partial \mathbf{h}(t_0)}, \boxed{\frac{\partial L}{\partial \theta}}$ discussed later
  - The first one is actually an **adjoint state** at a(t=0)
  - We know adjoint state at $a(t=t_{end})$ since this is just a gradient of loss w.r.t. final hidden state
  - We can use adjoint state dynamics equation and integrate it to find a(t=0)
  - This can be done using exactly the same solver applied to forward dynamics

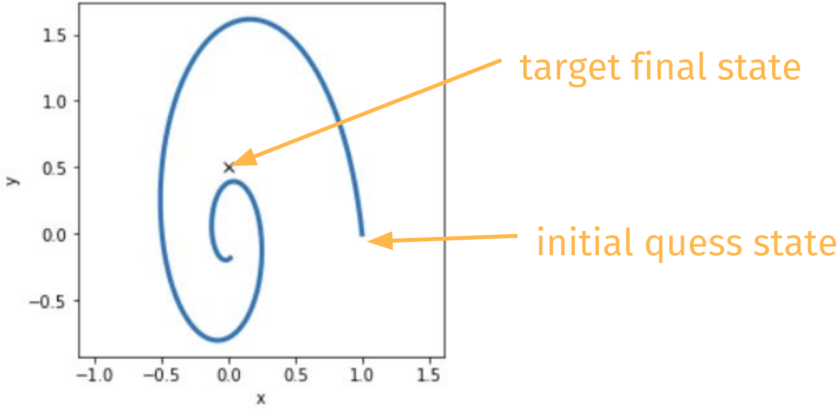
$$\mathbf{a}(t=0) = \mathbf{a}(t=t_{\text{end}}) - \int_{t=t_{\text{end}}}^{t=0} dt' \mathbf{a}(t') \frac{\partial f(\mathbf{h}(t'), t')}{\partial \mathbf{h}(t')}$$

NeuralODE - adjoint method - example usage

- Let as consider following toy problem:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t); \mathbf{W}) = \mathbf{W}\mathbf{h}(t) \qquad \mathbf{W} = \begin{pmatrix} -0.1 & 1 \\ -0.2 & -0.1 \end{pmatrix}$$

- We want to find such `h(t=0)` that `h(t=25)=(0, 1/2), with` **`dt=1/200`**
- Select some initial conditions e.g. `h(t=0)=(1, 0)`
- Integrate ODE: `h_hist = odeint(func, h0, time_steps)`



Implementation can be found in: **1A.Adjoint_method.ipynb**

NeuralODE - adjoint method - example

- Define cost function

$$\mathcal{L} = \left\| \mathbf{h}^{\text{final}} - \mathbf{h}^{\text{target}} \right\|^2 = \sum_{\text{k}=1}^{2} \left(\text{h}_{\text{k}}^{\text{final}} - \text{h}_{\text{k}}^{\text{target}} \right)^2$$

- Compute initial value for adjoint state (gradient of loss w.r.t last state):

$$\mathbf{a}(t=25) = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{\text{final}}} = 2\left(\mathbf{h}^{\text{final}} - \mathbf{h}^{\text{target}}\right)$$

- Solve adjoint state dynamics ODE by integrating backward in time i.e. from t=25 to t=0

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)\frac{\partial f(\mathbf{h}(t), t)}{\partial \mathbf{h}(t)} = -\mathbf{a}(t)\,\mathbf{W}$$

```
a_hist = odeint(lambda t, a: -matmul(a, W), 2*(hN - hT), t=t[::-1])
```

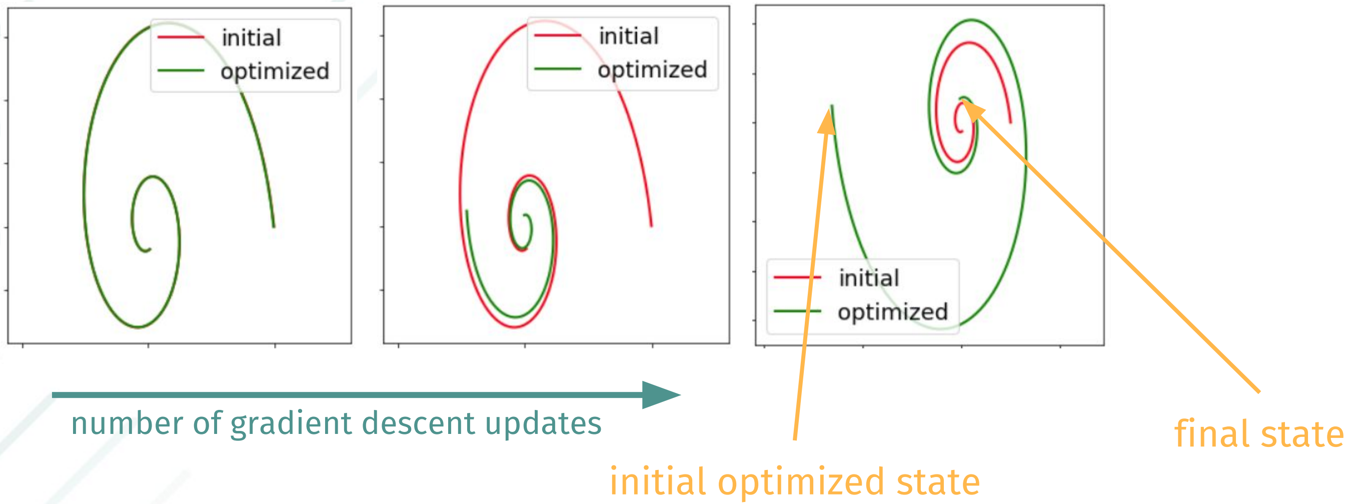
- a(t=0) is a gradient of dL/h(t=0) which we can apply to reduce the loss

NeuralODE - adjoint method - example

- update initial state with gradient direction

$$\mathbf{h}^{\text{initial}} := \mathbf{h}^{\text{initial}} - \lambda \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{\text{initial}}} = \mathbf{h}^{\text{initial}} - \lambda \mathbf{a}\,(t=0)$$

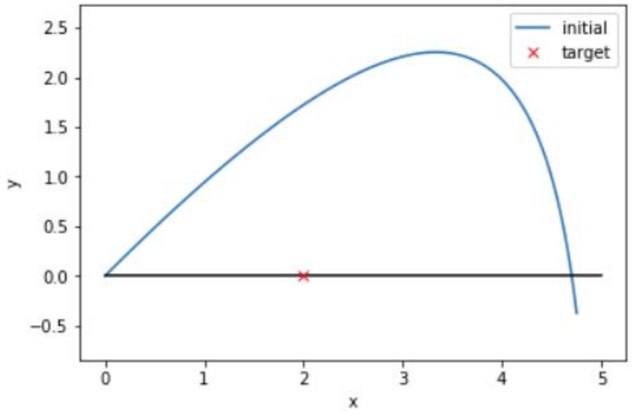
- Repeat steps the process until converge ...



Note: this problem can be solved by easily by inverting time in original problem and start problem from the final state :)

NeuralODE - adjoint method - example #2 (bullet problem)

- **bullet initial velocity problem:** given a target position x_t - estimate initial conditions for bullet velocity such that it will hit the target
- this problem cannot be easily solved by simply inverting dynamics since we don't know final velocity

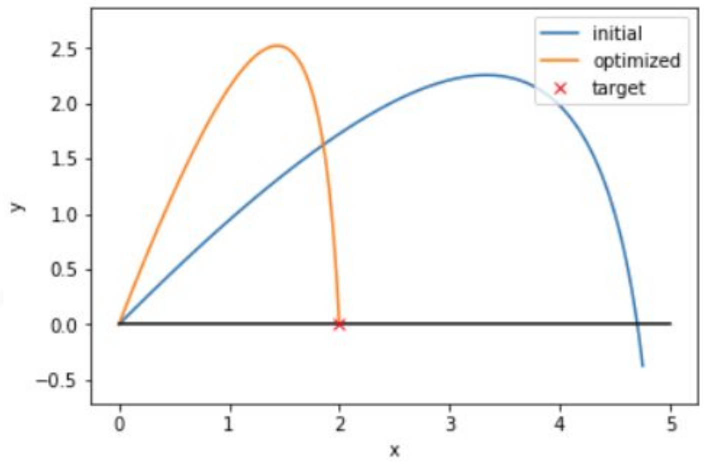


bullet trajectory for sample initial condition

- **Assumptions:** cannon is fixed at position **x=0** and we can change velocity vector v_0 its angle and magnitude (two parameters)

NeuralODE - adjoint method - another example

- the way we solve this problem is exactly the same
- we have to define cost and compute gradients w.r.t initial velocity i.e. the value of adjoint state **a**(t=0)



Implementation can be found in: **2.Demo_optimize_bullet_trajectory.ipynb**

NeuralODE - computing gradients w.r.t model parameters

- From last few slides we get set of formulas:

$$\frac{\mathrm{d}\mathbf{h}(\mathrm{t})}{\mathrm{dt}} = \mathrm{f}(\mathbf{h}(\mathrm{t}), \mathrm{t})$$

forward dynamics - some Neural Network

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{h}(t)}$$

adjoint state - a definition

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t) \frac{\partial f(\mathbf{h}(t), t)}{\boxed{\partial \mathbf{h}(t)}}$$

adjoint state dynamics - proved

- **How to compute gradients w.r.t model parameters ?**

It is easy to show that the adjoint equation for **weights** is (for an explanation check paper):

$$\frac{d\mathbf{a}_\theta(t)}{dt} = -\mathbf{a}(t) \frac{\partial f(\mathbf{h}(t), t)}{\partial \theta}$$

$$\frac{\partial \mathcal{L}}{\partial \theta}(t=0) = \underbrace{\mathbf{a}_\theta(t=t_{\text{end}})}_{=0} - \int_{t=t_{\text{end}}}^{t=0} dt' \mathbf{a}(t) \frac{\partial f(\mathbf{h}(t'), t')}{\boxed{\partial \theta}}$$

NeuralODE - final algorithm

- The final algorithm for the reverse mode computation (copied from paper):

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

---

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient ${\partial L}/{\partial \mathbf{z}(t_1)}$

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$): ▷ Define dynamics on augmented state

  return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

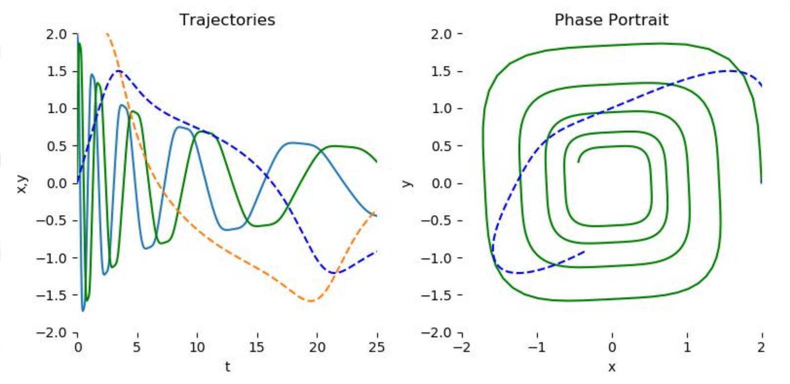
return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

---

- note that **input/output** of aug_dynamics fn can be expressed in terms of single vector which is expected by many ODE solvers implementations
- **ODESolve** is same as **odeint** function in previous slides
- whole implementation can fit ~100 lines of **python** code

NeuralODE - example usage of my implementation in TF

- The spiral problem (authors example)



1. generate target trajectory

```
t_grid = np.linspace(0, 25, data_size)
true_y0 = tf.to_float([[2., 0.]])
true_A = tf.to_float([[-0.1, 2.0], [-2.0, -0.1]])
class Lambda(tf.keras.Model):
    def call(self, inputs, **kwargs):
        t, r = inputs
        return tf.matmul(r**3, true_A)
neural_ode = NeuralODE(Lambda(), t=t_grid)
yN, states_history = neural_ode.forward(true_y0, return_states="numpy")
```

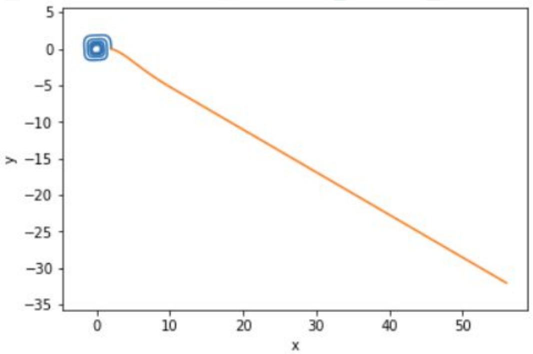
2. Model Neural Network

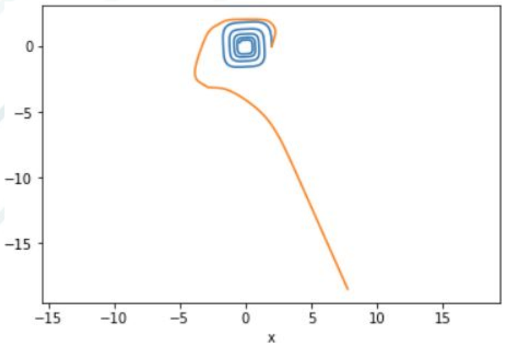
```
class ODEModel(tf.keras.Model):
    def __init__(self):
        super(ODEModel, self).__init__()
        self.linear1 = keras.layers.Dense(50, activation="tanh")
        self.linear2 = keras.layers.Dense(2)

    def call(self, inputs, **kwargs):
        t, y = inputs
        h = y**3
        h = self.linear1(h)
        h = self.linear2(h)
        return h
```

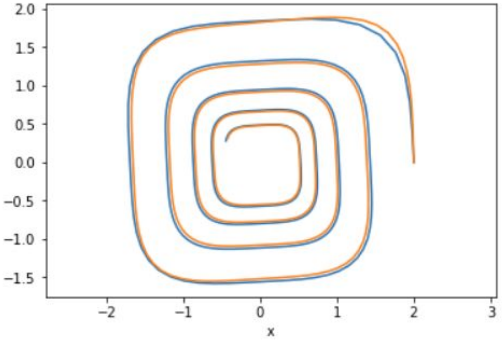
3. Use stochastic gradient descent to optimize target on:
`model = ODEModel()`

```
batch_y0, batch_yN = get_batch()
pred_y = neural_ode.forward(batch_y0)
with tf.GradientTape() as g:
    g.watch(pred_y)
    loss = tf.reduce_mean(tf.abs(pred_y - batch_yN))

dLoss = g.gradient(loss, pred_y)
h_start, dfdh0, dWeights = neural_ode.backward(pred_y, dLoss)
optimizer.apply_gradients(zip(dWeights, model.weights))
```

NeuralODE - spiral: results



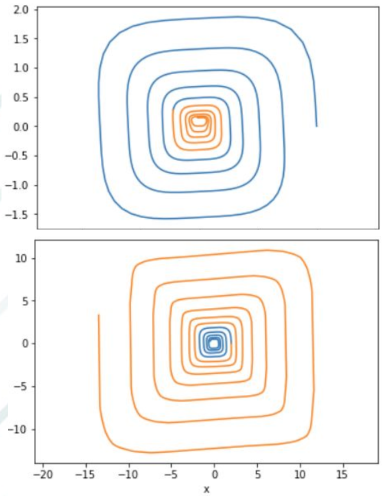




Initial trajectory

after few iterations

converged trajectory



We can do some extrapolation

```
true_yN = tf.to_float([true_y[-1]])
neural_ode_extrapolation =  NeuralODE(model, t = np.linspace(0, 500.0, data_size))
yN, states_history_model = neural_ode_extrapolation.forward(true_yN, return_states="numpy")
plot_spiral([true_y, np.concatenate(states_history_model)])
```

NeuralODE - MNIST experiment from paper

- Classification problem on MNIST dataset

Table 1: Performance on MNIST. †From LeCun et al. (1998).

	Test Error	# Params	Memory	Time
1-Layer MLP†	1.60%	0.24 M	-	-
ResNet	0.41%	0.60 M	$\mathcal{O}(L)$	$\mathcal{O}(L)$
RK-Net	0.47%	0.22 M	$\mathcal{O}(\tilde{L})$	$\mathcal{O}(\tilde{L})$
ODE-Net	0.42%	0.22 M	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(\tilde{L})$

- Less parameters while reaching similar accuracy

NeuralODE - MNIST experiment from paper

- Classification problem on MNIST dataset: properties of the adaptive solver

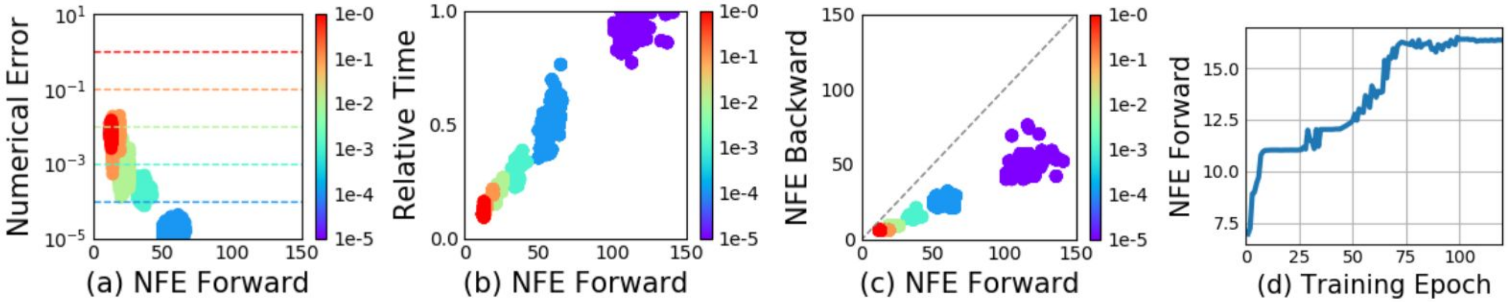


Figure 3: Statistics of a trained ODE-Net. (NFE = number of function evaluations.)

Figure 3c) shows a surprising result: the number of evaluations in the backward pass is roughly half of the forward pass. This suggests that the adjoint sensitivity method is not only more memory efficient, but also more computationally efficient than directly backpropagating through the integrator, because the latter approach will need to backprop through each function evaluation in the forward pass.

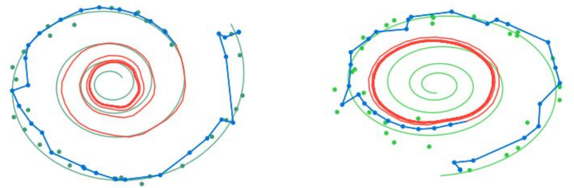
NeuralODE - A generative latent function time-series model

- We can train a generative model which will produce continuous latent vectors

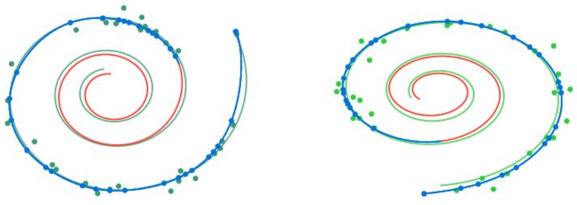
Appendix E Algorithm for training the latent ODE model

To obtain the latent representation $\mathbf{z}_{t_0}$, we traverse the sequence using RNN and obtain parameters of distribution $q(\mathbf{z}_{t_0}|\{\mathbf{x}_{t_i}, t_i\}_i, \theta_{enc})$. The algorithm follows a standard VAE algorithm with an RNN variational posterior and an ODESolve model:

1. Run an RNN encoder through the time series and infer the parameters for a posterior over $\mathbf{z}_{t_0}$:
$$q(\mathbf{z}_{t_0}|\{\mathbf{x}_{t_i}, t_i\}_i, \phi) = \mathcal{N}(\mathbf{z}_{t_0}|\mu_{\mathbf{z}_{t_0}}, \sigma_{\mathbf{z}_0}), \tag{53}$$
where $\mu_{\mathbf{z}_0}, \sigma_{\mathbf{z}_0}$ comes from hidden state of RNN($\{\mathbf{x}_{t_i}, t_i\}_i, \phi$)
2. Sample $\mathbf{z}_{t_0} \sim q(\mathbf{z}_{t_0}|\{\mathbf{x}_{t_i}, t_i\}_i)$
3. Obtain $\mathbf{z}_{t_1}, \mathbf{z}_{t_2}, \dots, \mathbf{z}_{t_M}$ by solving ODE ODESolve$(\mathbf{z}_{t_0}, f, \theta_f, t_0, \dots, t_M)$, where f is the function defining the gradient $d\mathbf{z}/dt$ as a function of $\mathbf{z}$
4. Maximize ELBO $= \sum_{i=1}^{M} \log p(\mathbf{x}_{t_i}|\mathbf{z}_{t_i}, \theta_{\mathbf{x}}) + \log p(\mathbf{z}_{t_0}) - \log q(\mathbf{z}_{t_0}|\{\mathbf{x}_{t_i}, t_i\}_i, \phi)$, where $p(\mathbf{z}_{t_0}) = \mathcal{N}(0, 1)$



(a) Recurrent Neural Network



(b) Latent Neural Ordinary Differential Equation

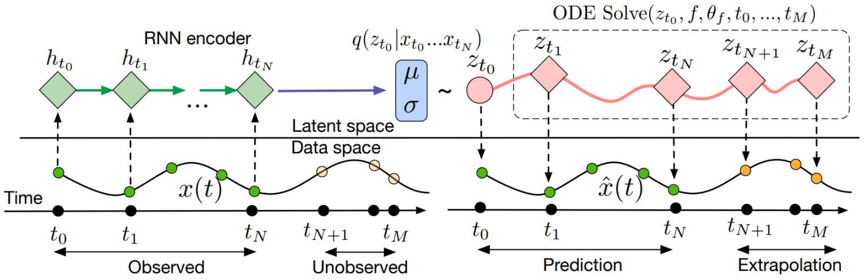


Figure 6: Computation graph of the latent ODE model.

NeuralODE - Tensorflow implementation of reverse mode

- Implementation of the reverse mode requires Jacobian-vector products

$$s_0 = \left[\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}\right]$$

$$\textbf{def}\ \text{aug\_dynamics}([\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta):$$

$$\quad \textbf{return}\ \left[f(\mathbf{z}(t), t, \theta), \boxed{-\mathbf{a}(t)^{\mathsf{T}}} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^{\mathsf{T}} \frac{\partial f}{\partial \theta}\right]$$

- Many frameworks do not support direct Jacobian computation or it’s inefficient, so we cannot easily compute **df/dz** and **df/dTheta** matrices above

- However, we can efficiently compute Jacobian vector products (**JVPs**) by providing **output_gradients** to gradient function, consider: **y** = L(**W*x**)=L(**h**)

- General reverse mode can be implemented in ~10 lines of code

```
def reverse_mode(self, t, hidden, ajdoint):
    ajdoint = -ajdoint

    with tf.GradientTape() as g:
        g.watch(hidden)
        hidden_out = self._model(inputs=[t, hidden])

    sources = [hidden] + self._model.weights
    gradients = g.gradient(
        target=hidden_out, sources=sources,
        output_gradients=ajdoint  # provide adjoint vector
    )
    # return [f(t, h), df/dHidden df/dTheta]
    return [hidden_out, *gradients]
```

Advanced part: Continuous Normalizing Flows

NeuralODE - Continuous Normalizing Flows (CNFs)

- Normalizing Flows (NFs) provide set of tools for computing probability distribution of transformed stochastic variable

The discretized equation (1) also appears in normalizing flows (Rezende and Mohamed, 2015) and the NICE framework (Dinh et al., 2014). These methods use the change of variables theorem to compute exact changes in probability if samples are transformed through a bijective function f:

$$\mathbf{z}_1 = f(\mathbf{z}_0) \implies \log p(\mathbf{z}_1) = \log p(\mathbf{z}_0) - \log \left| \det \frac{\partial f}{\partial \mathbf{z}_0} \right| \tag{6}$$

An example is the planar normalizing flow (Rezende and Mohamed, 2015):

$$\mathbf{z}(t+1) = \mathbf{z}(t) + uh(w^{\mathsf{T}}\mathbf{z}(t) + b), \quad \log p(\mathbf{z}(t+1)) = \log p(\mathbf{z}(t)) - \log \left| 1 + u^{\mathsf{T}} \frac{\partial h}{\partial \mathbf{z}} \right| \tag{7}$$

- For more details see on NFs my previous presentations

NeuralODE - Continuous Normalizing Flows (CNFs)

- Transformations in regular NFs:

$$\begin{cases} \mathbf{z}(t+1) & = f(\mathbf{z}(t), t) \\ \log p(\mathbf{z}(t+1)) & = \log p(\mathbf{z}(t)) - \log \left| \det \frac{\partial f(\mathbf{z}(t),t)}{\partial \mathbf{z}(t)} \right| \end{cases}$$

- CNFs analog (note **f** and **g** will be different functions):

$$\begin{cases} \frac{d\mathbf{z}}{dt} & = g(\mathbf{z}(t), t) \\ \frac{\partial \log p(\mathbf{z}(t))}{\partial t} & = -\mathrm{tr}\left(\frac{\partial g(\mathbf{z}(t),t)}{\partial \mathbf{z}(t)} \right) \end{cases}$$

proof in paper

- Computing trace is more efficient than computing determinant $O(N^3)$
- However computing jacobian is not efficient, we can approximate trace by sampling, see [FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models]

NeuralODE - CNFs implementation

- More detail on my Github repo

- We can still use the same API as in case of NeuralODE
- t, z_p_concat = inputs
- **hyper_net(t)** returns NFs parameters in function of time
- compute f(z(t), t)

$$\frac{d\mathbf{z}(t)}{dt} = uh(w^{\mathsf{T}}\mathbf{z}(t) + b)$$

- compute jacobian

$$\frac{\partial \log p(\mathbf{z}(t))}{\partial t} = -u^{\mathsf{T}} \boxed{\frac{\partial h}{\partial \mathbf{z}(t)}}$$

- concat dz/dt with dlogpz to have one vector: ode solver accepts single vector

```
class CNF(tf.keras.Model):

    def __init__(self, input_dim, hidden_dim, n_ensemble):
        super().__init__()
        self.hyper_net = HyperNet(input_dim, hidden_dim, n_ensemble)

    def call(self, inputs, **kwargs):
        t, z_p_concat = inputs
        z = z_p_concat[:, :self.hyper_net.input_dim]
        W, B, U = self.hyper_net(t)
        Z = tf.tile(tf.expand_dims(z, 0), [self.hyper_net.n_ensemble, 1, 1])

        with tf.GradientTape() as g:
            g.watch(Z)
            h = tf.tanh(tf.matmul(Z, W) + B)
            dzdt = tf.reduce_mean(tf.matmul(h, U), 0)
            reduced_h = tf.reduce_sum(h)

        dhdZ = g.gradient(
            target=reduced_h,
            sources=Z,
        )
        dlogpz = -tf.matmul(dhdZ, tf.transpose(U, [0, 2, 1]))
        dlogpz = tf.reduce_mean(dlogpz, axis=0)
        return tf.concat([dzdt, dlogpz], axis=1)
```

NeuralODE - CNFs results

- CNFs are naturally trained with **Maximum Likelihood Method**

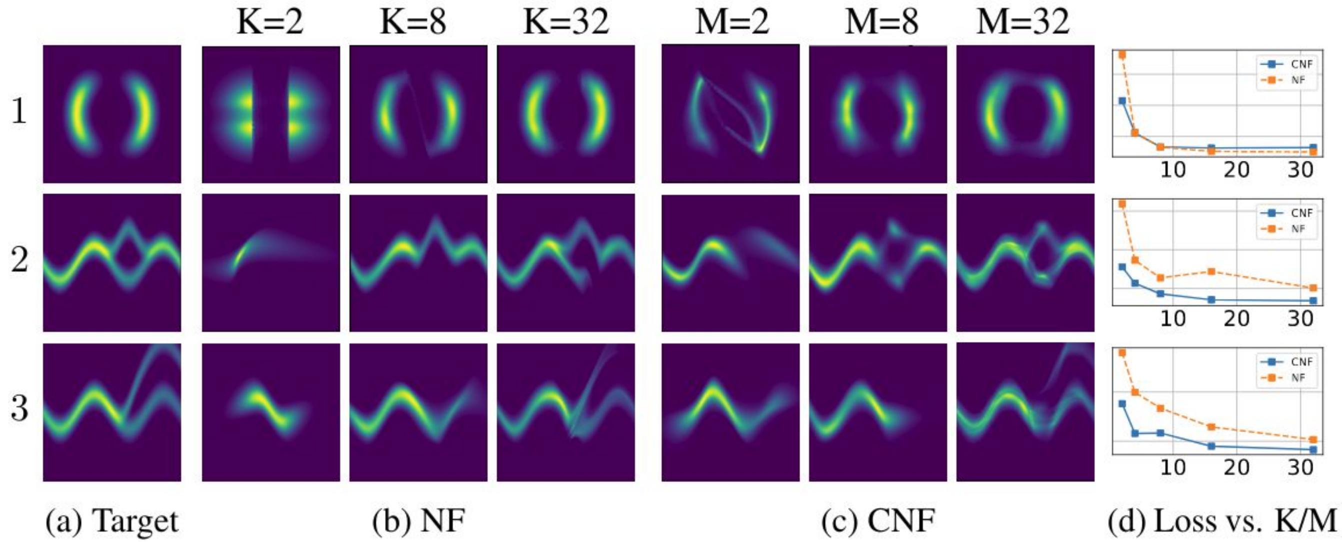


Figure 4: Comparison of normalizing flows versus continuous normalizing flows. The model capacity of normalizing flows is determined by their depth (K), while continuous normalizing flows can also increase capacity by increasing width (M), making them easier to train.

NeuralODE - CNFs results

- CNFs are invertible hence we can sample from them like in NFs
- Density evolution at different time snapshots
- Bottom rows show NFs results

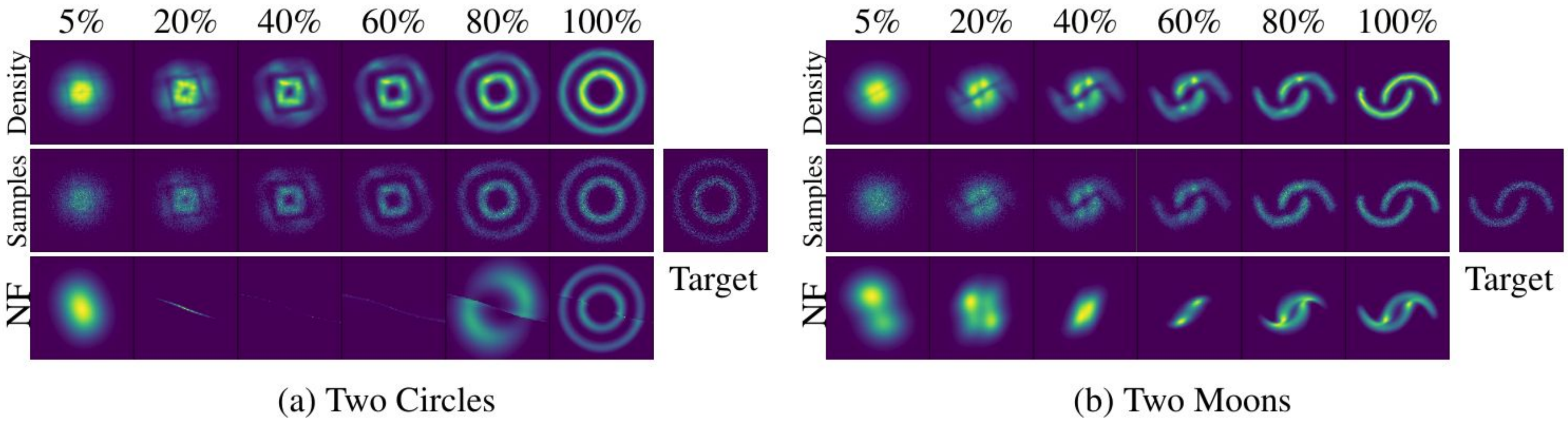


Figure 5: **Visualizing the transformation from noise to data.** Continuous-time normalizing flows are reversible, so we can train on a density estimation task and still be able to sample from the learned density efficiently.

NeuralODE - CNFs my experiments

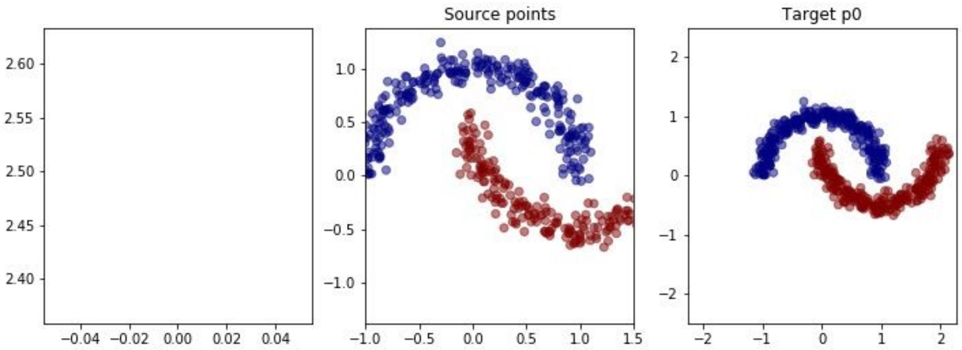
- Transforming two moons into unit gaussian in few lines

```
cnf_net = CNF(input_dim=2, hidden_dim=32, n_ensemble=16)
ode = NeuralODE(model=cnf_net, t=np.linspace(0, 1, 10))
```

```
# sample points from two moons dataset
x0 = tf.to_float(make_moons(n_samples=num_samples, noise=0.08)[0])
logdet0 = tf.zeros([num_samples, 1])
h0 = tf.concat([x0, logdet0], axis=1)
```

```
def compute_gradients_and_update(h0):
    hN = ode.forward(inputs=h0)
    with tf.GradientTape() as g:
        g.watch(hN)
        xN, logdetN = hN[:, :2], hN[:, 2]
        # L = log(p(zN))
        mle = tf.reduce_sum(p0.log_prob(xN), -1)
        # normally we maximize: log(p(z)) - logdetJ
        loss = - tf.reduce_mean(mle - logdetN)

    dloss = g.gradient(loss, hN)
    h0_rec, dLdh0, dLdW = ode.backward(hN, dloss)
    optimizer.apply_gradients(zip(dLdW, cnf_net.weights))
    return loss
```

Z transformations to unit gaussian during training:

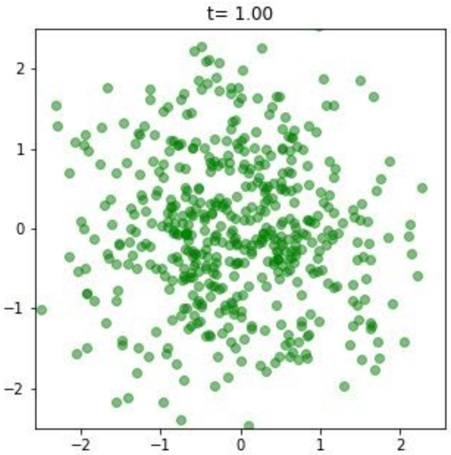


NeuralODE - CNFs my experiments

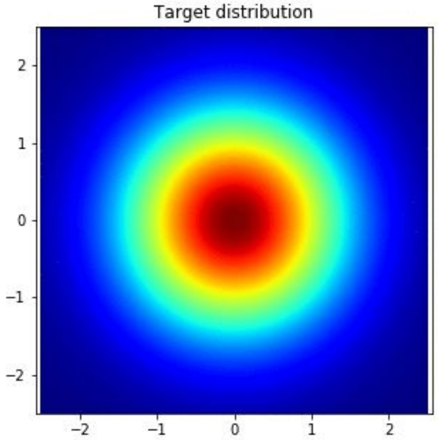
- Sampling from trained model (more detail in my notebooks)

```python
cnf_net = CNF(input_dim=2, hidden_dim=32, n_ensemble=16)
ode = NeuralODE(model=cnf_net, t=np.linspace(0, 1, 10))
```

```python
p0 = tf.distributions.Normal(loc=[0.0, 0.0], scale=[1.0, 1.0])
hN_sample = tf.concat([p0.sample(num_samples), logdet0], axis=1)
h0_sample, *_ = ode.backward(outputs=hN_sample)
```



- t=1 corresponds to initial state: we start from unit gaussian
- t=0 should generate moon distribution
- (**right->**) continuous transformation of density function



More animations here: youtube

Solving dynamical systems with ODEs - summary

NeuralODE - summary

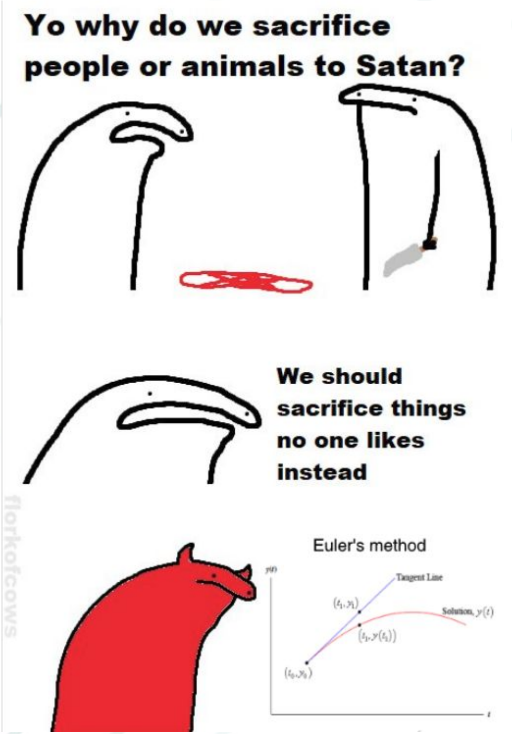
- NeuralODEs are new type of Neural Networks,
- They are defined in terms of dynamical ODE

$$\frac{d\mathbf{h}(t)}{dt} = \mathrm{NNetwork}\left(\mathbf{h}(t), t, \theta\right)$$

- The number of layers can be related with number of ODE solver steps
- Adaptive solvers allow to reduce number of steps at cost of lower precision, but higher speed
- So far NeuralODEs are slower than regular NNs of the same depth
- NeuralODEs are invertible and can be used to sample trajectories

References

- Replacing Neural Networks with Black-Box ODE Solvers - slides from presentation
- Reddit discussion on TF implementation
- Understanding Neural ODE's - blog post on Neural ODEs
- Neural Ordinary Differential Equations and Adversarial Attacks - blog post with some tests with Adversarial attacks on NeuralODEs
- FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models - a follow up paper of the authors which focus mainly on CNFs, has nice trick how to approximate jacobian trace via sampling
- Notes on Adjoint Methods for 18.335 - as in title notes on adjoint method
- Neural Ordinary Differential Equations - reference paper


Yo why do we sacrifice people or animals to Satan?
We should sacrifice things no one likes instead
Euler's method
Tangent Line
Solution, y(t)
florkofcows

Thank you!

NeuralODE - Continuous Normalizing Flows (CNFs)

- We can apply the same approach as in case of regular NNs and consider continuous transformation of variables
- In case of planar flows we have:

$$\mathbf{z}(t+1) = \mathbf{z}(t) + \mathbf{u}\tanh\left(\underbrace{\mathbf{w}^T\mathbf{z}(t) + b}_{\text{scalar}}\right)$$

- A continuous analog of it:

$$\frac{d\mathbf{z}}{dt} = \mathbf{u}\tanh\left(\mathbf{w}^T\mathbf{z}(t) + b\right) = f(\mathbf{z}(t), t)$$

Note: In CNFs f(z, t) can be non-bijective

- What is the dynamics of probability distribution ?

Theorem 1 (Instantaneous Change of Variables). *Let $\mathbf{z}(t)$ be a finite continuous random variable with probability $p(\mathbf{z}(t))$ dependent on time. Let $\frac{d\mathbf{z}}{dt} = f(\mathbf{z}(t), t)$ be a differential equation describing a continuous-in-time transformation of $\mathbf{z}(t)$. Assuming that f is uniformly Lipschitz continuous in $\mathbf{z}$ and continuous in t, then the change in log probability also follows a differential equation,*

$$\frac{\partial \log p(\mathbf{z}(t))}{\partial t} = -\mathrm{tr}\left(\frac{df}{d\mathbf{z}(t)}\right) \tag{8}$$